МИНИСТЕРСТВО ТРАНСПОРТА РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ   
«РОССИЙСКИЙ УНИВЕРСИТЕТ ТРАНСПОРТА»**

**РУТ (МИИТ)**

**Кафедра «Управление и защита информации»**

КУРСОВОЙ ПРОЕКТ

по дисциплине

«Системы искусственого интеллекта»

на тему:

*«Решение задачи бинарной классификации:прогноз риска развития сердечно-сосудистых заболеваний»*

Выполнил: студент гр. УВП-412 Рогов К.Д.

Принял: доцент Зольникова Н. Н.

Москва – 2025

ОГЛАВЛЕНИЕ

[ВВЕДЕНИЕ 3](#_Toc188377742)

[1 Предварительная обработка данных и постановка задачи 4](#_Toc188377743)

[1.1 Описание набора данных 4](#_Toc188377744)

[1.2 Подготовка данных 4](#_Toc188377745)

[2 Реализация бинарной классификации двумя способами 6](#_Toc188377746)

[2.1 Модель без использования библиотек 6](#_Toc188377747)

[2.2 Модель с использованием библиотек 9](#_Toc188377748)

[3 Разработка программы для работы с обученной моделью 11](#_Toc188377749)

[Заключение 14](#_Toc188377750)

[ПРИЛОЖЕНИЕ А 15](#_Toc188377751)

# **ВВЕДЕНИЕ**

В данной курсовой работе решается задача бинарной классификации на реальных данных о факторах риска сердечно-сосудистых заболеваний (согласно варианту, выбранному совместно с преподавателем в Задании 1). Основная цель — разработать систему искусственного интеллекта (СИИ), которая будет прогнозировать вероятность развития сердечно-сосудистого заболевания (в течение ближайших 10 лет) на основании ряда признаков: пола, возраста, уровня образования, количества выкуриваемых сигарет, показателей давления и т.д.

Особенность проекта состоит в том, что требуется реализовать решение задачи двумя способами:

1. Реализация всех алгоритмов «вручную» (без использования высокоуровневых библиотек машинного обучения).
2. Реализация с применением стандартных библиотек (в частности, scikit-learn).

Затем необходимо выполнить сравнение полученных результатов (точность, F1-мера и др.) и проанализировать, насколько «ручная» реализация совпадает (или отличается) от библиотечных методов.

В завершение нужно написать небольшое приложение (GUI), демонстрирующее процесс работы с уже обученной моделью: пользователь вводит значения 15 признаков, после чего приложение выводит результат предсказания (есть риск либо его нет).

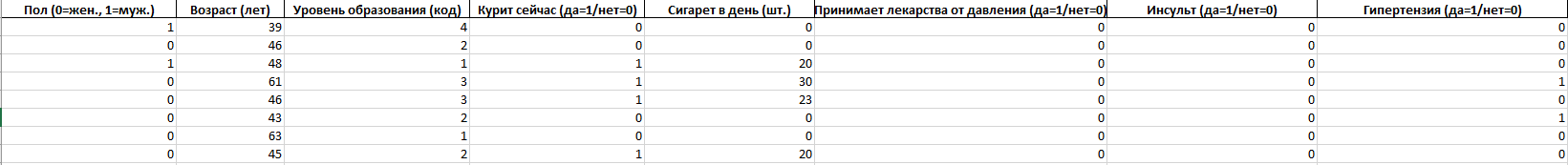
# **1 Предварительная обработка данных и постановка задачи**

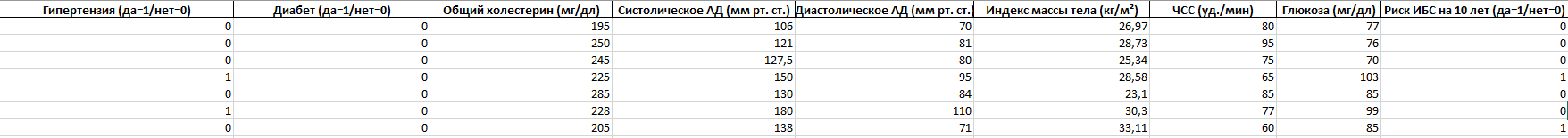
## Описание набора данных

Набор данных содержит сведения о пациентах и различных факторах риска (переменные, связанные с развитием сердечно-сосудистых заболеваний). Для каждого пациента в выборке присутствуют следующие признаки:

* Числовые (возраст, количество сигарет в день, общий холестерин, систолическое и диастолическое давление, индекс массы тела, частота пульса, глюкоза).
* Категориальные (пол, уровень образования, факт курения, приём препаратов от давления, наличие инсульта в анамнезе, гипертония, диабет).

Целевая переменная (TenYearCHD) принимает значение 1, если риск сердечно-сосудистого заболевания в течение 10 лет высок, и 0 — в обратном случае (нет риска).





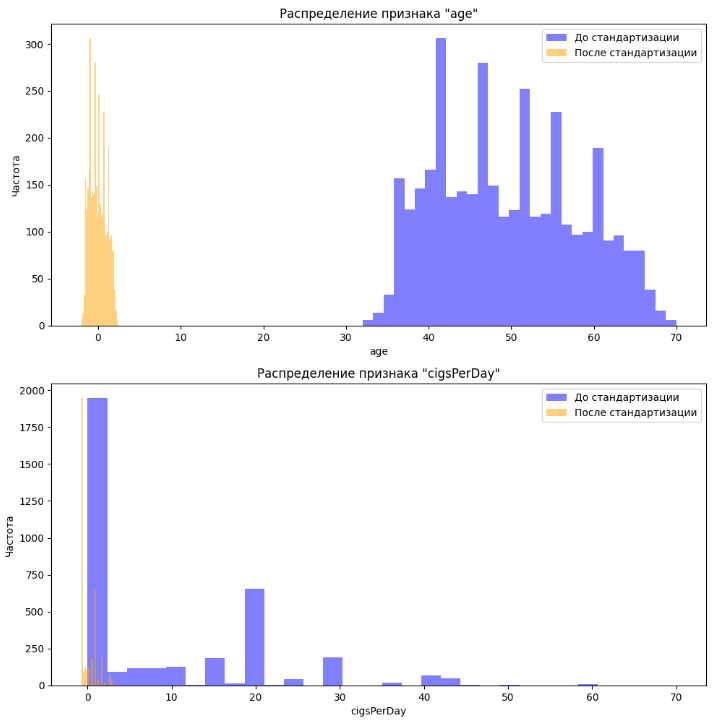
## Подготовка данных

Этапы подготовки данных

1. **Удаление пропусков**: при наличии строк с неполными данными они были исключены.
2. **Стандартизация признаков**: к числовым полям применён **Z-score** (вычитание среднего и деление на стандартное отклонение), что обеспечивает равный вклад всех признаков в обучение модели.
3. **Перемешивание (shuffle)**: для корректного обучения и снижения риска переобучения.
4. **Разбиение на обучающую и тестовую выборки**: ~75% данных под обучение и ~25% под тестирование.

Ниже приведён важный **фрагмент кода** (Python), показывающий, как выполняются загрузка, очистка, стандартизация и сохранение обработанного набора данных:

# Определяем функцию для перемешивания данных  
def shuffle\_data(dataframe, seed=42):  
 *"""  
 Перемешивает строки DataFrame вручную для воспроизводимости результатов.  
 """* # Устанавливаем фиксированное значение seed для воспроизводимости  
 random.seed(seed)  
 # Преобразуем данные в список списков для удобства перестановок  
 lines = dataframe.values.tolist()  
 # Алгоритм перемешивания: для каждого элемента выбираем случайный индекс и меняем элементы местами  
 for i in range(len(lines) - 1, 0, -1):  
 j = random.randint(0, i)  
 lines[i], lines[j] = lines[j], lines[i]  
 # Преобразуем данные обратно в DataFrame с исходными столбцами  
 return pd.DataFrame(lines, columns=dataframe.columns)  
  
def preprocess\_and\_save():  
 *"""  
 1) Загружает ../dataset.csv с использованием pandas.  
 2) Удаляет строки с пропущенными значениями.  
 3) Применяет стандартизацию Z-score к числовым признакам.  
 4) Перемешивает набор данных.  
 5) Делит данные на обучающую и тестовую выборки и сохраняет их в файлы:  
 - ../normalized\_shuffled\_train.csv  
 - ../normalized\_shuffled\_test.csv  
 6) Сохраняет график сравнения распределений в ../standardization\_comparison\_all\_features.png  
 """* # ------------------ Загрузка набора данных ------------------ #  
 original\_dataset = pd.read\_csv('../dataset.csv')  
  
 # Удаление строк с любыми пропущенными значениями  
 original\_dataset = original\_dataset.dropna()  
  
 # Определение числовых и категориальных признаков  
 numerical\_features = [  
 'age', 'cigsPerDay', 'totChol', 'sysBP',  
 'diaBP', 'BMI', 'heartRate', 'glucose'  
 ]  
 categorical\_features = [  
 'male', 'education', 'currentSmoker', 'BPMeds',  
 'prevalentStroke', 'prevalentHyp', 'diabetes'  
 ]  
  
 # Создаем копию данных для стандартизации  
 dataset = original\_dataset.copy()  
  
 # --------------- Применяем стандартизацию Z-score --------------- #  
 means = {}  
 scales = {}  
 for feature in numerical\_features:  
 mean = dataset[feature].mean()  
 std = dataset[feature].std()  
 means[feature] = mean  
 scales[feature] = std  
 dataset[feature] = (dataset[feature] - mean) / std  
  
 print("Сохранение параметров масштабирования в файл:", 'scaler\_params.pkl')  
 with open('scaler\_params.pkl', "wb") as f:  
 pickle.dump({"means": means, "scales": scales}, f)  
  
 # ------------------ Перемешивание набора данных ------------------ #  
 dataset = shuffle\_data(dataset, seed=42)  
  
 # --------------- Разделение набора данных на обучающую и тестовую выборки --------------- #  
 train\_size = int(0.75 \* len(dataset))  
 train\_data = dataset.iloc[:train\_size]  
 test\_data = dataset.iloc[train\_size:]

На графиках показано распределение числовых признаков до и после стандартизации. Видно, что исходные значения имеют различный диапазон, тогда как после нормализации их распределение центрируется вокруг 0, что является важным шагом для корректного обучения модели.

# **Реализация бинарной классификации двумя способами**

## 2.1 Модель без использования библиотек

В качестве «ручного» алгоритма для решения задачи бинарной классификации была выбрана **логистическая регрессия** с обучением путём градиентного спуска.

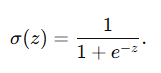
Логистическая регрессия — один из базовых алгоритмов машинного обучения для решения задач бинарной классификации. Несмотря на слово «регрессия» в названии, фактически речь идёт о прогнозировании вероятности принадлежности объекта к одному из двух классов (0 или 1). Основная идея:

1. Считается так называемый линейный скор (или «логит»):



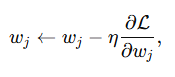
Здесь w0— свободный член (bias, смещение), а wi— веса, соответствующие каждому признаку xi ​.

1. К линейной части применяется сигмоидная функция σ(z), переводящая значение z в диапазон от 0 до 1:



В результате сигмоид даёт нам оценку вероятности того, что объект относится к классу «1» при имеющихся входных признаках.

1. Критерий, который обычно оптимизирует логистическая регрессия, — логарифмическая функция потерь (log loss или бинарная кросс-энтропия). Однако в учебных целях часто используют среднеквадратичную ошибку (MSE), чтобы наглядно показывать процесс «спуска» и смотреть, уменьшается ли ошибка. С точки зрения классической теории логистической регрессии логарифмическая функция потерь чаще даёт более корректные результаты, но MSE тоже можно использовать, особенно для демонстрации.
2. Градиентный спуск (Gradient Descent) — итеративный метод оптимизации, который двигает вектор весов w в направлении, противоположном градиенту функции потерь. Формально, на каждом шаге мы делаем обновление:



где η— скорость обучения (learning rate), а L — функция потерь (в нашем случае MSE). При корректном выборе η и достаточном количестве эпох обучение сходится.

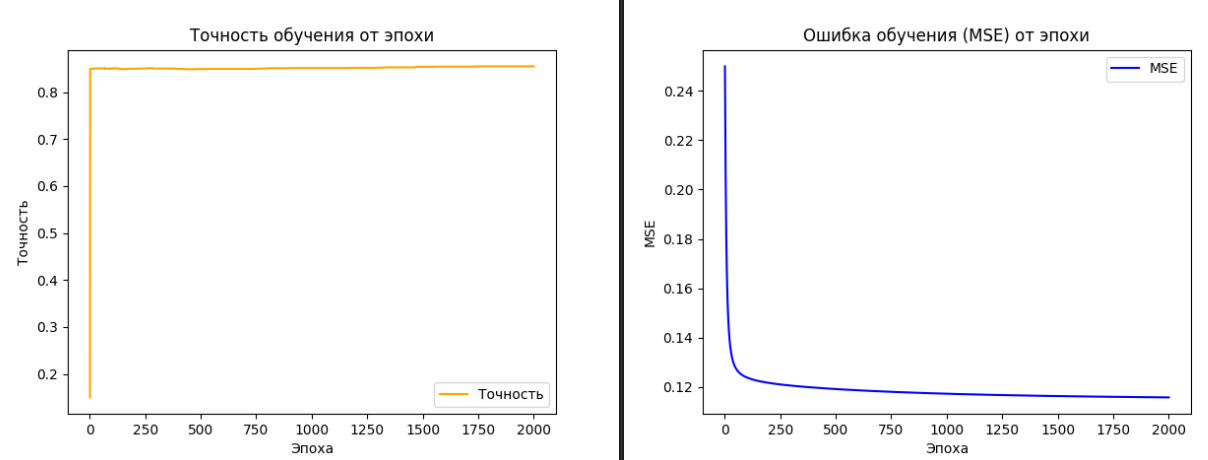
В коде, показанном ниже, вся логика — от вычисления сигмоиды до подсчёта производной — написана вручную. Это позволяет лучше понять внутреннее устройство алгоритма.

def sigmoid(z):  
 return 1.0 / (1.0 + math.exp(-z))  
  
def train\_manual(X, y, epochs=2000, learning\_rate=0.05):  
 *"""  
 Выполняет градиентный спуск для логистической регрессии вручную.  
 Возвращает:  
 weights: итоговые веса (смещение + коэффициенты)  
 errors: список значений MSE на каждую эпоху  
 accuracies: список точностей на каждую эпоху  
 """* # Добавляем столбец смещения: форма -> (n\_samples, n\_features+1)  
 X = [[1.0] + x for x in X]  
  
 # Инициализация весов  
 weights = [0.0] \* len(X[0])  
  
 errors = []  
 accuracies = []  
  
 for epoch in range(epochs):  
 # Прямой проход  
 predictions = []  
 for row in X:  
 z = sum(w \* x for w, x in zip(weights, row))  
 predictions.append(sigmoid(z))  
  
 # MSE для отслеживания  
 error = sum((p - t) \*\* 2 for p, t in zip(predictions, y)) / len(y)  
 errors.append(error)  
  
 # Точность  
 predicted\_labels = [1 if p >= 0.5 else 0 for p in predictions]  
 accuracy = sum(1 for pl, t in zip(predicted\_labels, y) if pl == t) / len(y)  
 accuracies.append(accuracy)  
  
 # Расчет градиента  
 for j in range(len(weights)):  
 grad = sum((p - t) \* row[j] for p, t, row in zip(predictions, y, X)) / len(y)  
 weights[j] -= learning\_rate \* grad  
  
 return weights, errors, accuracies

После обучения мы оцениваем итоговые метрики вручную : **Точность** (Accuracy) ~ 0.83

График ошибки : Ошибка MSE постепенно уменьшается с увеличением количества эпох, показывая процесс сходимости модели.

График точности : Точность модели достигает стабильного значения около 83,7%.



## 2.2 Модель с использованием библиотек

Второй способ — применение библиотеки **scikit-learn**. Аналогично выбираем **логистическую регрессию**, но обучаем её через SGDClassifier в итеративном режиме.

Вместо того чтобы вручную писать сигмоиду и циклы обновления весов, можно использовать scikit-learn — популярную библиотеку для машинного обучения на Python. Класс SGDClassifier:

* Реализует стохастический градиентный спуск (Stochastic Gradient Descent) или его вариации.
* Параметр loss='log\_loss' указывает на использование логистической регрессии.
* learning\_rate='constant' + eta0=learning\_rate даёт нам фиксированную скорость обучения.
* max\_iter=1 + warm\_start=True позволяют пошагово вызывать partial\_fit и самим контролировать число эпох (2000).

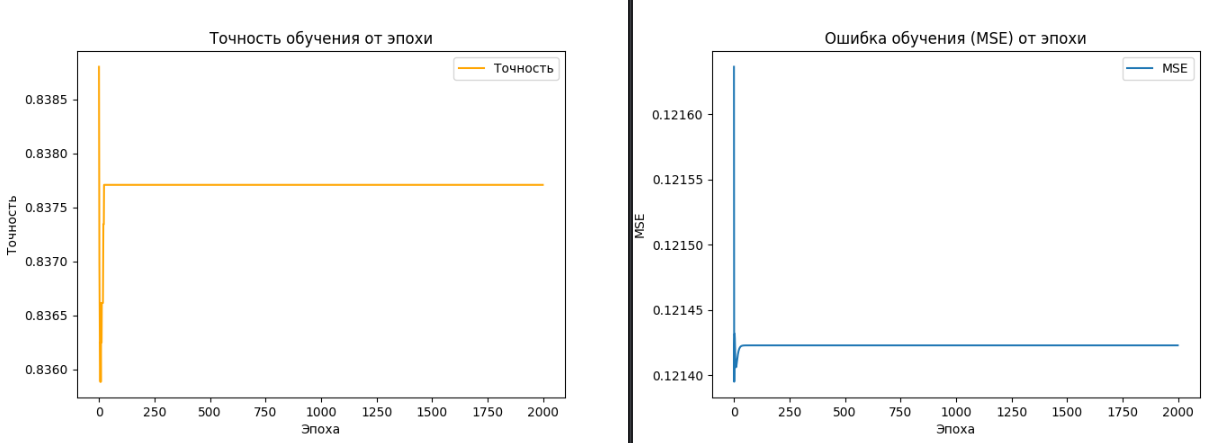
Ключевые отличия от «ручной» реализации:

1. Оптимизация вычислений: scikit-learn зачастую делает внутреннюю векторизацию, что даёт прирост в скорости и стабильности расчётов.
2. Log-loss по умолчанию\*\*: при loss='log\_loss' идёт оптимизация логистической функции потерь, которая обычно лучше согласуется с теорией логистической регрессии (хотя MSE мы можем тоже отслеживать для наглядности).
3. Различные режимы обновления: помимо partial\_fit, можно использовать «полный» вызов fit или mini-batch режимы.

# Функция обучения модели методом итеративного обучения  
def train\_model\_iterative(X, y, epochs=2000, learning\_rate=0.05):  
 *"""  
 Обучает модель логистической регрессии с использованием SGDClassifier в итеративном режиме.  
 """* model = SGDClassifier(  
 loss='log\_loss', # Логистическая регрессия  
 penalty=None, # Без регуляризации  
 learning\_rate='constant',  
 eta0=learning\_rate,  
 max\_iter=1, # Обучение в 1 эпоху за вызов partial\_fit  
 warm\_start=True, # Для многократного вызова partial\_fit  
 random\_state=0  
 )  
  
 # Преобразуем X и y в массивы numpy  
 X\_np = X.to\_numpy()  
 y\_np = y.to\_numpy()  
 errors = []  
 accuracies = []  
 # SGDClassifier требует список классов при первом вызове partial\_fit  
 classes = np.unique(y\_np)  
  
 for epoch in range(epochs):  
 # partial\_fit выполняет 1 эпоху обучения  
 model.partial\_fit(X\_np, y\_np, classes=classes)  
  
 # Прогнозы вероятностей для расчета MSE  
 pred\_probs = model.predict\_proba(X\_np)[:, 1]  
 mse = np.mean((pred\_probs - y\_np) \*\* 2)  
 errors.append(mse)  
  
 # Точность (accuracy)  
 y\_pred = model.predict(X\_np)  
 acc = accuracy\_score(y\_np, y\_pred)  
 accuracies.append(acc)  
  
 return model, errors, accuracies

График ошибки: Процесс обучения с библиотекой также показывает снижение MSE, что подтверждает корректность работы модели. Начальная ошибка ниже, а снижение более плавное. Использование SGDClassifier из scikit-learn позволяет автоматически настраивать вычисления и использовать более точные алгоритмы оптимизации. Это может объяснять лучшее начало обучения и более быстрый выход на оптимальную зону.

График точности: Результат аналогичен модели без библиотек, с точностью около 84%. Точность растет более плавно, особенно на первых итерациях. Оптимизации в библиотеке (SGDClassifier) позволяют быстрее и эффективнее достигать оптимальных значений, избегая скачков в обучении.

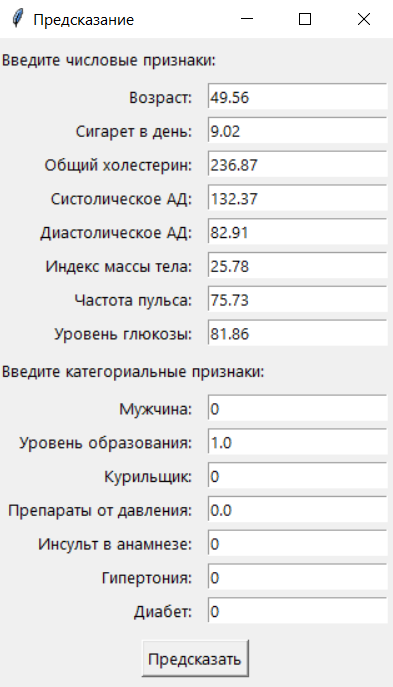


# **3 Разработка программы для работы с обученной моделью**

После того как модель обучена, возникает потребность дать пользователю удобный инструмент для ввода новых данных и получения результата. Для этих целей было реализовано **простое графическое приложение (GUI)** на базе библиотеки **Tkinter**.

При запуске **GUI** пользователь заполняет поля для всех 15 признаков (8 числовых и 7 категориальных). Числовые значения автоматически проходят через те же параметры стандартизации (средние и стандартные отклонения), которые были получены при обучении. Затем программа загружает ранее сохранённые веса model\_manual\_weights.txt и выполняет прогноз (0 или 1).

class HeartDiseasePredictorGUI:  
 def \_\_init\_\_(self, master):  
 self.master = master  
 self.master.title("Предсказание")  
  
 self.entries = {}  
 row\_index = 0  
  
 # Числовые признаки  
 tk.Label(master, text="Введите числовые признаки:").grid(row=row\_index, column=0, columnspan=2, pady=5, sticky="w")  
 row\_index += 1  
 for feat, label in numerical\_features.items():  
 tk.Label(master, text=f"{label}:").grid(row=row\_index, column=0, padx=5, pady=2, sticky="e")  
 entry = tk.Entry(master)  
 entry.insert(0, f"{means.get(feat, 0):.2f}")  
 entry.grid(row=row\_index, column=1, padx=5, pady=2)  
 self.entries[feat] = entry  
 row\_index += 1  
  
 # Категориальные признаки  
 tk.Label(master, text="Введите категориальные признаки:").grid(row=row\_index, column=0, columnspan=2, pady=5, sticky="w")  
 row\_index += 1  
 for feat, label in categorical\_features.items():  
 tk.Label(master, text=f"{label}:").grid(row=row\_index, column=0, padx=5, pady=2, sticky="e")  
 entry = tk.Entry(master)  
 entry.insert(0, f"{modes.get(feat, 0)}") # Используем моду  
 entry.grid(row=row\_index, column=1, padx=5, pady=2)  
 self.entries[feat] = entry  
 row\_index += 1  
  
 # Кнопка предсказания  
 self.predict\_button = tk.Button(master, text="Предсказать", command=self.on\_predict)  
 self.predict\_button.grid(row=row\_index, column=0, columnspan=2, pady=10)  
  
 row\_index += 1  
 self.result\_label = tk.Label(master, text="", font=("Helvetica", 12, "bold"))  
 self.result\_label.grid(row=row\_index, column=0, columnspan=2, pady=10)  
  
 self.manual\_weights = load\_manual\_weights("../models/model\_manual\_weights.txt")  
  
 def on\_predict(self):  
 # Проверяем ввод  
 for key, entry in self.entries.items():  
 if not entry.get().strip():  
 self.result\_label.config(text="Заполните все поля!", fg="red")  
 return  
 # Считываем ввод  
 user\_input\_numeric = [float(self.entries[feat].get()) for feat in numerical\_features]  
 user\_input\_categorical = [float(self.entries[feat].get()) for feat in categorical\_features]  
  
 # Стандартизируем числовые признаки  
 standardized\_numeric = [(val - means.get(feat, 0)) / scales.get(feat, 1) for val, feat in zip(user\_input\_numeric, numerical\_features)]  
  
 all\_inputs = standardized\_numeric + user\_input\_categorical  
 # Предсказание  
 prediction = predict\_manual(all\_inputs, self.manual\_weights)  
 msg = "Результат: риск есть (1)" if prediction == 1 else"Результат:риска нет (0)"  
 self.result\_label.config(text=msg, fg="green")



# **Заключение**

В рамках данной курсовой работы была реализована система искусственного интеллекта, решающая задачу бинарной классификации (предсказание сердечно-сосудистого риска). Ключевые результаты и выводы:

1. Подготовка данных: удаление пропусков, стандартизация, разбиение на обучающую и тестовую выборки.
2. Два подхода к обучению:
   * «Ручная» реализация логистической регрессии на Python (поэтапный градиентный спуск).
   * Использование библиотеки scikit-learn (SGDClassifier) для обучения той же модели.
3. Сравнение результатов: обе модели показывают сопоставимую точность ~84%, незначительные расхождения связаны с нюансами реализации.
4. Создание GUI: разработано приложение на Tkinter, в котором пользователь может ввести новые данные и получить предсказание риска (0 или 1).

Таким образом, проект демонстрирует понимание основ машинного обучения, навыки программирования, работу с библиотеками, а также умение внедрять результат в удобное пользовательское приложение.

# **ПРИЛОЖЕНИЕ А**

import random  
import pickle  
import pandas as pd  
import matplotlib.pyplot as plt  
  
# Определяем функцию для перемешивания данных  
def shuffle\_data(dataframe, seed=42):  
 *"""  
 Перемешивает строки DataFrame вручную для воспроизводимости результатов.  
 """* # Устанавливаем фиксированное значение seed для воспроизводимости  
 random.seed(seed)  
 # Преобразуем данные в список списков для удобства перестановок  
 lines = dataframe.values.tolist()  
 # Алгоритм перемешивания: для каждого элемента выбираем случайный индекс и меняем элементы местами  
 for i in range(len(lines) - 1, 0, -1):  
 j = random.randint(0, i)  
 lines[i], lines[j] = lines[j], lines[i]  
 # Преобразуем данные обратно в DataFrame с исходными столбцами  
 return pd.DataFrame(lines, columns=dataframe.columns)  
  
def preprocess\_and\_save():  
 *"""  
 1) Загружает ../dataset.csv с использованием pandas.  
 2) Удаляет строки с пропущенными значениями.  
 3) Применяет стандартизацию Z-score к числовым признакам.  
 4) Перемешивает набор данных.  
 5) Делит данные на обучающую и тестовую выборки и сохраняет их в файлы:  
 - ../normalized\_shuffled\_train.csv  
 - ../normalized\_shuffled\_test.csv  
 6) Сохраняет график сравнения распределений в ../standardization\_comparison\_all\_features.png  
 7) Сохраняет параметры масштабирования и моды категориальных признаков в scaler\_params.pkl  
 """* # ------------------ Загрузка набора данных ------------------ #  
 original\_dataset = pd.read\_csv('../dataset.csv')  
  
 # Удаление строк с любыми пропущенными значениями  
 original\_dataset = original\_dataset.dropna()  
  
 # Определение числовых и категориальных признаков  
 numerical\_features = [  
 'age', 'cigsPerDay', 'totChol', 'sysBP',  
 'diaBP', 'BMI', 'heartRate', 'glucose'  
 ]  
 categorical\_features = [  
 'male', 'education', 'currentSmoker', 'BPMeds',  
 'prevalentStroke', 'prevalentHyp', 'diabetes'  
 ]  
  
 # Создаем копию данных для стандартизации  
 dataset = original\_dataset.copy()  
  
 # --------------- Применяем стандартизацию Z-score --------------- #  
 means = {}  
 scales = {}  
 for feature in numerical\_features:  
 mean = dataset[feature].mean()  
 std = dataset[feature].std()  
 means[feature] = mean  
 scales[feature] = std  
 dataset[feature] = (dataset[feature] - mean) / std  
  
 # Вычисляем моды для категориальных признаков  
 modes = {}  
 for feature in categorical\_features:  
 modes[feature] = dataset[feature].mode()[0]  
  
 print("Сохранение параметров масштабирования и мод в файл:", 'scaler\_params.pkl')  
 with open('scaler\_params.pkl', "wb") as f:  
 pickle.dump({"means": means, "scales": scales, "modes": modes}, f)  
  
 # ------------------ Перемешивание набора данных ------------------ #  
 dataset = shuffle\_data(dataset, seed=42)  
  
 # --------------- Разделение набора данных на обучающую и тестовую выборки --------------- #  
 train\_size = int(0.75 \* len(dataset))  
 train\_data = dataset.iloc[:train\_size]  
 test\_data = dataset.iloc[train\_size:]  
  
 # --------------- Сохранение обработанных данных --------------- #  
 train\_data.to\_csv('../normalized\_shuffled\_train.csv', index=False)  
 test\_data.to\_csv('../normalized\_shuffled\_test.csv', index=False)  
  
 # --------------- Визуализация: сравнение распределений --------------- #  
 fig, axes = plt.subplots(len(numerical\_features), 1, figsize=(10, 5 \* len(numerical\_features)))  
  
 for idx, feature in enumerate(numerical\_features):  
 axes[idx].hist(original\_dataset[feature], bins=30, alpha=0.5, label='До стандартизации', color='blue')  
 axes[idx].hist(dataset[feature], bins=30, alpha=0.5, label='После стандартизации', color='orange')  
 axes[idx].set\_title(f'Распределение признака "{feature}"')  
 axes[idx].set\_xlabel(feature)  
 axes[idx].set\_ylabel('Частота')  
 axes[idx].legend()  
  
 plt.tight\_layout()  
 plt.savefig('../standardization\_comparison\_all\_features.png')  
 plt.close()  
  
# Добавляем основной блок для выполнения функции  
if \_\_name\_\_ == "\_\_main\_\_":  
 preprocess\_and\_save()

import pickle  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
  
from sklearn.linear\_model import SGDClassifier  
from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score  
  
# Функция загрузки набора данных  
def load\_dataset(file\_path):  
 *"""  
 Загружает CSV файл с набором данных по указанному пути.  
 """* try:  
 dataset = pd.read\_csv(file\_path)  
 print(f"Набор данных успешно загружен: {dataset.shape[0]} строк и {dataset.shape[1]} столбцов.")  
 return dataset  
 except Exception as e:  
 print(f"Ошибка при загрузке набора данных: {e}")  
 raise  
  
# Функция для разделения признаков и целевой переменной  
def split\_features\_target(dataset, target\_column):  
 *"""  
 Разделяет набор данных на признаки (X) и целевую переменную (y).  
 """* X = dataset.drop(columns=[target\_column])  
 y = dataset[target\_column]  
 return X, y  
  
# Функция обучения модели методом итеративного обучения  
def train\_model\_iterative(X, y, epochs=2000, learning\_rate=0.05):  
 *"""  
 Обучает модель логистической регрессии с использованием SGDClassifier в итеративном режиме.  
 """* model = SGDClassifier(  
 loss='log\_loss', # Логистическая регрессия  
 penalty=None, # Без регуляризации  
 learning\_rate='constant',  
 eta0=learning\_rate,  
 max\_iter=1, # Обучение в 1 эпоху за вызов partial\_fit  
 warm\_start=True, # Для многократного вызова partial\_fit  
 random\_state=0  
 )  
  
 # Преобразуем X и y в массивы numpy  
 X\_np = X.to\_numpy()  
 y\_np = y.to\_numpy()  
  
 errors = []  
 accuracies = []  
  
 # SGDClassifier требует список классов при первом вызове partial\_fit  
 classes = np.unique(y\_np)  
  
 for epoch in range(epochs):  
 # partial\_fit выполняет 1 эпоху обучения  
 model.partial\_fit(X\_np, y\_np, classes=classes)  
  
 # Прогнозы вероятностей для расчета MSE  
 pred\_probs = model.predict\_proba(X\_np)[:, 1]  
 mse = np.mean((pred\_probs - y\_np) \*\* 2)  
 errors.append(mse)  
  
 # Точность (accuracy)  
 y\_pred = model.predict(X\_np)  
 acc = accuracy\_score(y\_np, y\_pred)  
 accuracies.append(acc)  
  
 return model, errors, accuracies  
  
# Функция оценки модели  
def evaluate\_model(model, X, y):  
 *"""  
 Оценивает финальные метрики на всем наборе данных (в данном случае тренировочном).  
 """* y\_pred = model.predict(X)  
 metrics = {  
 "Точность": accuracy\_score(y, y\_pred),  
 "Точность (Precision)": precision\_score(y, y\_pred),  
 "Полнота (Recall)": recall\_score(y, y\_pred),  
 "F1-мера": f1\_score(y, y\_pred)  
 }  
 return metrics  
  
# Функция сохранения модели  
def save\_model(model, file\_path):  
 *"""  
 Сохраняет обученную модель в файл в формате pickle.  
 """* with open(file\_path, 'wb') as f:  
 pickle.dump(model, f)  
 print(f"Модель сохранена в файл: {file\_path}")  
  
# Функция построения графиков истории обучения  
def plot\_training\_history(errors, accuracies, error\_plot\_path, accuracy\_plot\_path):  
 *"""  
 Сохраняет два графика:  
 1) MSE от эпохи  
 2) Точность (accuracy) от эпохи  
 """* # График MSE от эпохи  
 plt.figure(figsize=(7, 5))  
 plt.plot(range(len(errors)), errors, label='MSE')  
 plt.title('Ошибка обучения (MSE) от эпохи')  
 plt.xlabel('Эпоха')  
 plt.ylabel('MSE')  
 plt.legend()  
 plt.savefig(error\_plot\_path)  
 plt.close()  
  
 # График точности от эпохи  
 plt.figure(figsize=(7, 5))  
 plt.plot(range(len(accuracies)), accuracies, label='Точность', color='orange')  
 plt.title('Точность обучения от эпохи')  
 plt.xlabel('Эпоха')  
 plt.ylabel('Точность')  
 plt.legend()  
 plt.savefig(accuracy\_plot\_path)  
 plt.close()  
  
# Основной блок выполнения программы  
if \_\_name\_\_ == "\_\_main\_\_":  
 dataset\_path = '../../datasets/normalized\_shuffled\_train.csv'  
 model\_save\_path = '../model\_lib\_weights.pkl'  
  
 error\_plot\_path = '../sklearn\_error\_plot.png'  
 accuracy\_plot\_path = '../sklearn\_accuracy\_plot.png'  
  
 target\_column = 'TenYearCHD'  
 EPOCHS = 2000  
 LR = 0.05  
  
 # Шаг 1: Загрузка набора данных  
 dataset = load\_dataset(dataset\_path)  
  
 # Шаг 2: Разделение на признаки и целевую переменную  
 X\_train, y\_train = split\_features\_target(dataset, target\_column)  
  
 # Шаг 3: Обучение модели  
 print("Начало обучения модели...")  
 model, errors, accuracies = train\_model\_iterative(X\_train, y\_train, EPOCHS, LR)  
  
 # Шаг 4: Построение графиков  
 plot\_training\_history(errors, accuracies, error\_plot\_path, accuracy\_plot\_path)  
  
 # Шаг 5: Оценка модели  
 metrics = evaluate\_model(model, X\_train, y\_train)  
 print("Итоговые метрики обучения:", metrics)  
  
 # Шаг 6: Сохранение модели  
 save\_model(model, model\_save\_path)  
 print("Обучение завершено.")

import math  
  
# Функция для загрузки набора данных вручную  
def load\_dataset(file\_path):  
 *"""  
 Читает CSV-файл построчно и преобразует значения в числа с плавающей точкой.  
 """* with open(file\_path, 'r') as file:  
 lines = file.readlines()  
  
 headers = lines[0].strip().split(',')  
 data = [list(map(float, line.strip().split(','))) for line in lines[1:]]  
 return headers, data  
  
def sigmoid(z):  
 return 1.0 / (1.0 + math.exp(-z))  
  
def train\_manual(X, y, epochs=2000, learning\_rate=0.05):  
 *"""  
 Выполняет градиентный спуск для логистической регрессии вручную.  
 Возвращает:  
 weights: итоговые веса (смещение + коэффициенты)  
 errors: список значений MSE на каждую эпоху  
 accuracies: список точностей на каждую эпоху  
 """* # Добавляем столбец смещения: форма -> (n\_samples, n\_features+1)  
 X = [[1.0] + x for x in X]  
  
 # Инициализация весов  
 weights = [0.0] \* len(X[0])  
  
 errors = []  
 accuracies = []  
  
 for epoch in range(epochs):  
 # Прямой проход  
 predictions = []  
 for row in X:  
 z = sum(w \* x for w, x in zip(weights, row))  
 predictions.append(sigmoid(z))  
  
 # MSE для отслеживания  
 error = sum((p - t) \*\* 2 for p, t in zip(predictions, y)) / len(y)  
 errors.append(error)  
  
 # Точность  
 predicted\_labels = [1 if p >= 0.5 else 0 for p in predictions]  
 accuracy = sum(1 for pl, t in zip(predicted\_labels, y) if pl == t) / len(y)  
 accuracies.append(accuracy)  
  
 # Расчет градиента  
 for j in range(len(weights)):  
 grad = sum((p - t) \* row[j] for p, t, row in zip(predictions, y, X)) / len(y)  
 weights[j] -= learning\_rate \* grad  
  
 return weights, errors, accuracies  
  
def evaluate\_manual(weights, X, y):  
 *"""  
 Оценивает финальные метрики, используя полученные веса.  
 Возвращает словарь с метриками: точность, полнота, F1-мера.  
 """* # Добавляем столбец смещения  
 X = [[1.0] + x for x in X]  
  
 # Прогнозы  
 predictions = []  
 for row in X:  
 z = sum(w \* x for w, x in zip(weights, row))  
 predictions.append(sigmoid(z))  
  
 predicted\_labels = [1 if p >= 0.5 else 0 for p in predictions]  
  
 # Расчет метрик вручную  
 tp = sum(1 for pl, t in zip(predicted\_labels, y) if pl == 1 and t == 1)  
 fp = sum(1 for pl, t in zip(predicted\_labels, y) if pl == 1 and t == 0)  
 fn = sum(1 for pl, t in zip(predicted\_labels, y) if pl == 0 and t == 1)  
 tn = sum(1 for pl, t in zip(predicted\_labels, y) if pl == 0 and t == 0)  
  
 accuracy = (tp + tn) / len(y)  
 precision = tp / (tp + fp) if (tp + fp) > 0 else 0  
 recall = tp / (tp + fn) if (tp + fn) > 0 else 0  
 f1\_score = 2 \* (precision \* recall) / (precision + recall) if (precision + recall) > 0 else 0  
  
 return {  
 "Точность": accuracy,  
 "Полнота": recall,  
 "F1-мера": f1\_score,  
 "Прецизионность": precision  
 }  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 dataset\_path = '../../datasets/normalized\_shuffled\_train.csv'  
 weights\_save\_path = '../model\_manual\_weights.txt'  
  
 EPOCHS = 2000  
 LR = 0.05  
  
 # Шаг 1: Загрузка набора данных  
 headers, data = load\_dataset(dataset\_path)  
  
 # Шаг 2: Определение индексов признаков и целевой переменной  
 target\_index = headers.index('TenYearCHD')  
 feature\_indices = [i for i in range(len(headers)) if i != target\_index]  
  
 # Подготовка X и y  
 X\_train = [[row[i] for i in feature\_indices] for row in data]  
 y\_train = [row[target\_index] for row in data]  
  
 # Шаг 3: Обучение модели (градиентный спуск)  
 print("Начало обучения модели с использованием градиентного спуска...")  
 weights, errors, accuracies = train\_manual(X\_train, y\_train, epochs=EPOCHS, learning\_rate=LR)  
  
 # Шаг 4: Сохранение весов модели  
 with open(weights\_save\_path, 'w') as f:  
 f.write(','.join(map(str, weights)))  
 print(f"Веса модели сохранены в: {weights\_save\_path}")  
  
 print("Создание графиков...")  
  
 # Создание графиков без использования импортов наверху  
 def plot\_graph(x, y, title, xlabel, ylabel, filepath, color='blue'):  
 import matplotlib.pyplot as plt  
 plt.figure()  
 plt.plot(x, y, color=color, label=ylabel)  
 plt.title(title)  
 plt.xlabel(xlabel)  
 plt.ylabel(ylabel)  
 plt.legend()  
 plt.savefig(filepath)  
 plt.close()  
  
 epochs\_range = list(range(1, EPOCHS + 1))  
  
 # График ошибки MSE  
 plot\_graph(  
 epochs\_range,  
 errors,  
 title='Ошибка обучения (MSE) от эпохи',  
 xlabel='Эпоха',  
 ylabel='MSE',  
 filepath='../manual\_model\_mse\_plot.png',  
 color='blue'  
 )  
  
 # График точности  
 plot\_graph(  
 epochs\_range,  
 accuracies,  
 title='Точность обучения от эпохи',  
 xlabel='Эпоха',  
 ylabel='Точность',  
 filepath='../manual\_model\_accuracy\_plot.png',  
 color='orange'  
 )  
  
 print("Графики успешно сохранены.")  
  
 # Шаг 5: Вывод итоговых метрик обучения  
 metrics = evaluate\_manual(weights, X\_train, y\_train)  
 print("Итоговые метрики обучения:")  
 for metric, value in metrics.items():  
 print(f"{metric}: {value:.4f}")  
  
 print("Обучение завершено.")

import pandas as pd  
import pickle  
import openpyxl  
from openpyxl.styles import PatternFill, Alignment  
  
  
def load\_dataset(file\_path):  
 with open(file\_path, 'r') as file:  
 lines = file.readlines()  
 headers = lines[0].strip().split(',')  
 data = [list(map(float, line.strip().split(','))) for line in lines[1:]]  
 return headers, data  
  
  
headers, data = load\_dataset('../../datasets/normalized\_shuffled\_test.csv')  
  
feature\_indices = [i for i in range(len(headers)) if headers[i] != 'TenYearCHD']  
target\_index = headers.index('TenYearCHD')  
X = [[row[i] for i in feature\_indices] for row in data]  
y = [row[target\_index] for row in data]  
  
with open('../model\_lib\_weights.pkl', 'rb') as f:  
 model\_lib = pickle.load(f)  
  
  
def load\_weights(file\_path):  
 with open(file\_path, 'r') as file:  
 weights = list(map(float, file.read().strip().split(',')))  
 return weights  
  
  
weights = load\_weights('../model\_manual\_weights.txt')  
  
  
def sigmoid(z):  
 return 1 / (1 + (2.71828 \*\* -z))  
  
  
def predict\_manual(X, weights):  
 X = [[1.0] + row for row in X]  
 predictions = []  
 for row in X:  
 z = sum(w \* x for w, x in zip(weights, row))  
 predictions.append(1 if sigmoid(z) >= 0.5 else 0)  
 return predictions  
  
  
predictions\_lib = model\_lib.predict(pd.DataFrame(X, columns=headers[:-1]))  
predictions\_manual = predict\_manual(X, weights)  
  
wb = openpyxl.Workbook()  
sheet = wb.active  
sheet.title = "Сравнение моделей"  
  
color\_both\_right = PatternFill(start\_color="90EE90", end\_color="90EE90", fill\_type="solid")  
color\_lib\_right = PatternFill(start\_color="ADD8E6", end\_color="ADD8E6", fill\_type="solid")  
color\_manual\_right = PatternFill(start\_color="FFFFE0", end\_color="FFFFE0", fill\_type="solid")  
color\_none\_right = PatternFill(start\_color="FFB6C1", end\_color="FFB6C1", fill\_type="solid")  
  
accuracy\_lib = sum([1 for i in range(len(y)) if predictions\_lib[i] == y[i]]) / len(y)  
accuracy\_manual = sum([1 for i in range(len(y)) if predictions\_manual[i] == y[i]]) / len(y)  
cases\_both\_correct = sum(1 for i in range(len(y)) if predictions\_lib[i] == y[i] and predictions\_manual[i] == y[i])  
cases\_lib\_only = sum(1 for i in range(len(y)) if predictions\_lib[i] == y[i] and predictions\_manual[i] != y[i])  
cases\_manual\_only = sum(1 for i in range(len(y)) if predictions\_manual[i] == y[i] and predictions\_lib[i] != y[i])  
cases\_neither = sum(1 for i in range(len(y)) if predictions\_lib[i] != y[i] and predictions\_manual[i] != y[i])  
  
sheet.append(["Точность (с библиотеками)", accuracy\_lib])  
sheet.append(["Точность (без библиотек)", accuracy\_manual])  
sheet.append(["обе модели предсказали верно", cases\_both\_correct / len(y)])  
sheet.append(["Предсказала верно модель с библиотеками", cases\_lib\_only / len(y)])  
sheet.append(["Предсказала верно модель без библиотек", cases\_manual\_only / len(y)])  
sheet.append(["Обе модели предсказали неверно", cases\_neither / len(y)])  
  
sheet.append([])  
  
sheet["A3"].fill = color\_both\_right  
sheet["A4"].fill = color\_lib\_right  
sheet["A5"].fill = color\_manual\_right  
sheet["A6"].fill = color\_none\_right  
  
sheet.append(headers + ["", "С библиотеками", "Без библиотек"])  
  
for cell in sheet[sheet.max\_row]:  
 cell.alignment = Alignment(horizontal="center")  
  
start\_row = sheet.max\_row + 1  
  
for i, row in enumerate(X):  
 actual = int(y[i])  
 pred\_lib = int(predictions\_lib[i])  
 pred\_manual = int(predictions\_manual[i])  
  
 if pred\_lib == actual and pred\_manual == actual:  
 fill = color\_both\_right  
 elif pred\_lib == actual and pred\_manual != actual:  
 fill = color\_lib\_right  
 elif pred\_manual == actual and pred\_lib != actual:  
 fill = color\_manual\_right  
 else:  
 fill = color\_none\_right  
  
 current\_row = start\_row + i  
 sheet.append(row + [actual, "", pred\_lib, pred\_manual])  
  
 for col in range(1, len(row) + 5):  
 sheet.cell(row=current\_row, column=col).fill = fill  
  
for col in sheet.columns:  
 values = [str(cell.value) for cell in col if cell.value is not None]  
 if values:  
 max\_length = max(len(value) for value in values)  
 adjusted\_width = max\_length + 2  
 sheet.column\_dimensions[col[0].column\_letter].width = adjusted\_width  
  
wb.save("../../results/model\_comparison.xlsx")

*"""  
Файл: gui\_predict.py  
  
Простое графическое приложение на Tkinter, которое:  
1) Запрашивает 15 признаков из вашего набора данных:  
 - 8 числовых признаков, требующих стандартизации  
 - 7 категориальных признаков (например, 0 или 1).  
2) Применяет ту же стандартизацию (используя сохраненные средние и стандартные отклонения).  
3) Загружает веса логистической регрессии из файла model\_manual\_weights.txt.  
4) Показывает предсказанный результат.  
"""*import tkinter as tk  
import pickle  
import math  
  
# 1) Определяем числовые и категориальные признаки с переводом.  
numerical\_features = {  
 'age': 'Возраст',  
 'cigsPerDay': 'Сигарет в день',  
 'totChol': 'Общий холестерин',  
 'sysBP': 'Систолическое АД',  
 'diaBP': 'Диастолическое АД',  
 'BMI': 'Индекс массы тела',  
 'heartRate': 'Частота пульса',  
 'glucose': 'Уровень глюкозы'  
}  
categorical\_features = {  
 'male': 'Мужчина',  
 'education': 'Уровень образования',  
 'currentSmoker': 'Курильщик',  
 'BPMeds': 'Препараты от давления',  
 'prevalentStroke': 'Инсульт в анамнезе',  
 'prevalentHyp': 'Гипертония',  
 'diabetes': 'Диабет'  
}  
all\_features = list(numerical\_features.keys()) + list(categorical\_features.keys())  
  
# 2) Загружаем параметры масштабирования.  
try:  
 scaler\_params = pickle.load(open("../datasets/datasets\_processing/scaler\_params.pkl", "rb"))  
 means = scaler\_params["means"]  
 scales = scaler\_params["scales"]  
 modes = scaler\_params["modes"] # Добавляем моды категориальных признаков  
except Exception as e:  
 print("Ошибка загрузки параметров масштабирования:", e)  
 means = {}  
 scales = {}  
 modes = {}  
  
# 3) Функция для загрузки весов.  
def load\_manual\_weights(file\_path):  
 with open(file\_path, 'r') as f:  
 weights\_str = f.read().strip().split(',')  
 return list(map(float, weights\_str))  
  
# 4) Сигмоида.  
def sigmoid(z):  
 return 1 / (1 + math.exp(-z))  
  
# 5) Предсказание.  
def predict\_manual(inputs, weights):  
 x\_with\_bias = [1.0] + inputs  
 z = sum(w \* x for w, x in zip(weights, x\_with\_bias))  
 return 1 if sigmoid(z) >= 0.5 else 0  
  
# 6) Графический интерфейс.  
class HeartDiseasePredictorGUI:  
 def \_\_init\_\_(self, master):  
 self.master = master  
 self.master.title("Предсказание")  
  
 self.entries = {}  
 row\_index = 0  
  
 # Числовые признаки  
 tk.Label(master, text="Введите числовые признаки:").grid(row=row\_index, column=0, columnspan=2, pady=5, sticky="w")  
 row\_index += 1  
 for feat, label in numerical\_features.items():  
 tk.Label(master, text=f"{label}:").grid(row=row\_index, column=0, padx=5, pady=2, sticky="e")  
 entry = tk.Entry(master)  
 entry.insert(0, f"{means.get(feat, 0):.2f}")  
 entry.grid(row=row\_index, column=1, padx=5, pady=2)  
 self.entries[feat] = entry  
 row\_index += 1  
  
 # Категориальные признаки  
 tk.Label(master, text="Введите категориальные признаки:").grid(row=row\_index, column=0, columnspan=2, pady=5, sticky="w")  
 row\_index += 1  
 for feat, label in categorical\_features.items():  
 tk.Label(master, text=f"{label}:").grid(row=row\_index, column=0, padx=5, pady=2, sticky="e")  
 entry = tk.Entry(master)  
 entry.insert(0, f"{modes.get(feat, 0)}") # Используем моду  
 entry.grid(row=row\_index, column=1, padx=5, pady=2)  
 self.entries[feat] = entry  
 row\_index += 1  
  
 # Кнопка предсказания  
 self.predict\_button = tk.Button(master, text="Предсказать", command=self.on\_predict)  
 self.predict\_button.grid(row=row\_index, column=0, columnspan=2, pady=10)  
  
 row\_index += 1  
 self.result\_label = tk.Label(master, text="", font=("Helvetica", 12, "bold"))  
 self.result\_label.grid(row=row\_index, column=0, columnspan=2, pady=10)  
  
 self.manual\_weights = load\_manual\_weights("../models/model\_manual\_weights.txt")  
  
 def on\_predict(self):  
 # Проверяем ввод  
 for key, entry in self.entries.items():  
 if not entry.get().strip():  
 self.result\_label.config(text="Заполните все поля!", fg="red")  
 return  
  
 # Считываем ввод  
 user\_input\_numeric = [float(self.entries[feat].get()) for feat in numerical\_features]  
 user\_input\_categorical = [float(self.entries[feat].get()) for feat in categorical\_features]  
  
 # Стандартизируем числовые признаки  
 standardized\_numeric = [(val - means.get(feat, 0)) / scales.get(feat, 1) for val, feat in zip(user\_input\_numeric, numerical\_features)]  
  
 all\_inputs = standardized\_numeric + user\_input\_categorical  
  
 # Предсказание  
 prediction = predict\_manual(all\_inputs, self.manual\_weights)  
  
 msg = "Результат: риск есть (1)" if prediction == 1 else "Результат: риска нет (0)"  
 self.result\_label.config(text=msg, fg="green")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 root = tk.Tk()  
 gui = HeartDiseasePredictorGUI(root)  
 root.mainloop()